1997

Improving Data Availability in Mobile Computing Using Prewrite Operations

Sanjay Kumar Madria

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Report Number:
97-032

IMPROVING DATA AVAILABLILITY IN MOBILE COMPUTING USING PREWRITE OPERATIONS

Sanjay Kumar Madria
Bharat Bhargava

CSD-TR 97-032
June 1997

Improving Data Availability In Mobile Computing Using Prewrite Operations[1]

Sanjay Kumar Madria
School of Computer Science
University Sains Malaysia
1800 Minden, Penang, Malaysia
skm@cs.usm.my

Bharat Bhargava
Department of Computer Sciences
Purdue University, West Lafayette
IN 47907, USA
bb@cs.purdue.edu

Abstract

Transactions consist of read and write operations. We incorporate a prewrite operation before a write operation in a mobile transaction to improve data availability. A prewrite operation does not update the state of a data object but only make visible the value the data object will have after the commit of the associated transaction. Once a transaction received all the values read and declares all the prewrites, it can pre-commit at mobile host (i.e., computer connected to unreliable communication) and the remaining transaction's execution is shifted to the stationary host (i.e., computer connected to the reliable fixed network). Writes on database after pre-commit take time and resources at stationary host and are therefore, delayed. This reduces network traffic congestion usually caused by updates. A pre-committed transaction's prewrite values are made visible both at mobile and stationary hosts before the final commit of the transaction. This increases data availability during frequent disconnection common in mobile computing. Since the expensive part of the transaction execution is shifted to the stationary host, it reduces the computing expenses (e.g., battery, low bandwidth, memory etc) at mobile host. Since a pre-committed transaction does not abort, no undo recovery needs to be performed in our model. A mobile host can cache only prewrite values of the data objects, which will take less space, time, energy and can be transmitted over low bandwidth.

1. Introduction

Technological advances in cellular communications, wireless LAN and satellite services have led to the emergence of mobile computing, also called nomadic computing. It is expected that in near future, tens of millions of users will carry a portable computer and communicator devices that uses a wireless connection to access a world wide information network ([IB],[PB3]). Wide area and wireless computing suggest that there will be more competition for shared data since it provide users with ability to access information and services through wireless connections that can be retained even while the user is moving. Further, mobile users will have to share their data with others. The task of ensuring consistency of share data becomes more difficult in mobile computing because of limitations of wireless communication channels and restrictions imposed due to mobility and portability. The access to the future information systems through mobile computers will be performed with the help of mobile transactions. However, a transaction in this environment is different than the transactions in the centralized or distributed databases in the following ways.

[1] This research is partly supported by a grant from NSF under NCR-9405931

- The mobile transactions might have to split their computations into sets of operations due to disconnection and mobility and may share their states and partial results with other transactions.
- The mobile transactions require computations and communications to be supported by stationary hosts. A mobile computations may be divided into a set of actions some of which execute on mobile host while others on stationary host.
- As the mobile hosts move from one cell to another, the states of transaction, states of accessed data objects, and the location information also move.
- The mobile transactions are long-lived transactions due to the mobility of both the data and users, and due to the frequent disconnections.
- The mobile transactions support and handle concurrency, recovery, disconnection and mutual consistency of the replicated data objects.

To support mobile transactions, the transaction processing models should accommodate the limitations of mobile computing, such as unreliable communication, limited battery life, low band-width communication, and reduced storage capacity. Mobile computations should minimize aborts due to disconnection. Operations on shared data must ensure correctness of transactions executed on both stationary and mobile hosts. The blocking of transactions execution on either the stationary or mobile hosts must be minimized to reduce communication cost and to increase concurrency. Proper support for mobile transactions must provide for local autonomy to allow transactions to be processed and committed on the mobile host despite temporary disconnection.

Our objective is to :

- increase data availability at mobile and stationary hosts. The system ensures serializability and ACID properties for mobile database applications.

The steps in our mobile transaction processing model are as follows: pre-read, read, prewrite, pre-commit and commit. These operations are explained below.

The main features of our mobile transaction model are :

- Each mobile transaction has a prewrite operation before a write operation. A prewrite operation makes visible the value the data object will have after the commit of the transaction.
- Once all the prewrites have been processed, the mobile transaction pre-commits at mobile host. A pre-committed transaction's results are visible at mobile and stationary hosts before the final commit. This minimizes the blocking of other transactions and increases concurrency.
- The transaction continues its execution at mobile host by announcing prewrite values and then shifts the resource consuming part of the transaction's execution (updates of the database on disk) to the stationary host. This reduces the computing cost on mobile host.
- A pre-committed transaction is guaranteed to commit. This feature of our model avoids an undo or compensating transaction, which is costly in mobile computing.
- A pre-read returns a prewrite value whereas a read returns a write value.

- The transactions are serialized based on their pre-commit order. We use two phase locking protocols to control concurrent conflicting operations. However, our scheme has optimistic nature.
- Our model deals efficiently with the constrained resources in mobile computing environment.

1.1 Mobile Architecture

In mobile computing environment (see figure 1), the network consists of stationary and mobile hosts [IB]. A **mobile host** (MH) changes its location and network connections while computations are being processed. While in motion, a mobile host retain its network connections through the support of stationary hosts with wireless connections. These stationary hosts are called **mobile support stations** (MSS) or base stations which perform the transaction and data management with the help of transaction managers (TMs) and data managers (DMs), respectively. Each MSS is responsible for all the mobile hosts within a given small geographical area, known as a cell. At any given instant, a MH communicates only with the MSS responsible for its cell. A MH may have some server capability to perform concurrency control and logging etc. Some MH has only an I/O capability. Within this mobile computing environment, shared data are expected to be stored and controlled by a number of database servers executing on an MSS.

When a MH leaves a cell serviced by a MSS, a **hand-off protocol** is used to transfer the responsibility for mobile transaction and data support to the MSS of the new cell. This hand-off may involve establishing a new communication link. It involves the migration of in progress transactions and database states from one MSS to another.

In mobile computing, there are several possible **modes of operations**. The operation mode may be fully connected (normal connection), totally disconnected (e.g., not a failure of MH) or partially connected (weak connection). Also, mobile computers may enter an energy conservation mode, called **doze state**. A doze state of MH does not imply the failure of the disconnected machine.

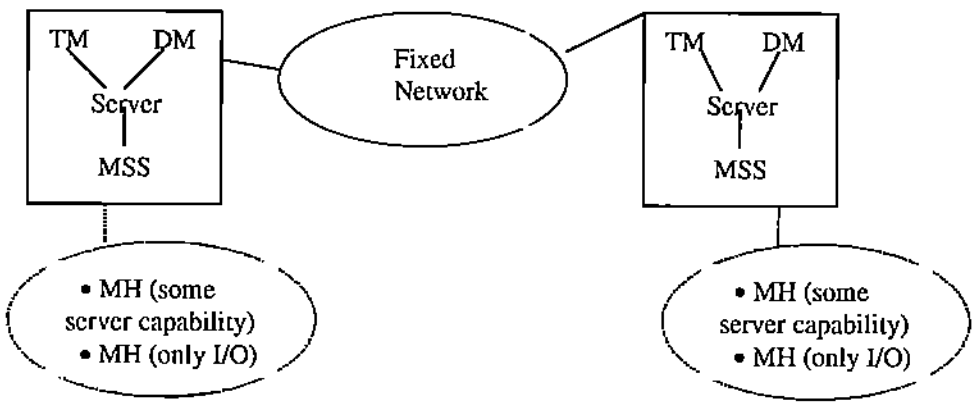

Figure 1. Mobile Architecture

1.2 Research Ideas in Mobile Transaction Processing

The mobile mobile transaction processing is an active area of research. We outline some of the ideas as follows.

- Semantic based transaction processing models ([BK],[RC]) have been extended for mobile computing in [WC] to increase concurrency by exploiting commutative operations. These techniques require caching large portion of the database or maintain multiple copies of many data items. In [WC], fragmentability of data objects have been used to facilitate semantic based transaction processing in mobile databases. The scheme fragments data objects. Each fragmented data object has to be cached independently and manipulated synchronously. This scheme works in the situations where the data objects can be fragmented like stacks and queues.
- In optimistic concurrency control based schemes [KS], cached objects on mobile hosts can be updated without any co-ordination but the updates need to be propagated and validated at the database servers for the commitment of transactions. This scheme leads to aborts of mobile transactions unless the conflicts are rare. Since mobile transactions are expected to be long-lived due to disconnection and long network delays, the conflicts will be more in mobile computing environment.
- In pessimistic schemes in which cached objects can be locked exclusively, mobile transactions can be committed locally. The pessimistic schemes lead to unnecessary transaction blocking since mobile hosts can not release any cached objects while it is disconnected. Existing caching methods attempt to cache the entire data objects or in some case the complete file. Caching of these potentially large objects over low bandwidth communication channels can result in wireless network congestion and high communication cost. The limited memory size of the MH allows only a small number of objects can be cached at any given time.
- Dynamic object clustering has been proposed in mobile computing in ([PB1],[PB2]) using weak-read, weak-write, strict-read and strict-write. Strict-read and strict-write have the same semantics as normal read and write operations invoked by transactions satisfying ACID properties [BHG]. A weak-read returns the value of a locally cached object written by a strict-write or a weak-write. A weak-write operation only updates a locally cached object, which might become permanent on cluster merging if the weak-write does not conflict with any strict-read or strict-write operation. The weak transactions use local and global commits. The 'local commit' is same as our 'pre-commit' and global commit is same as our final commit (see section 3). However, a weak transaction after local commit can abort and is compensated. In our model a pre-committed transaction does not abort, hence require no undo or compensation. A weak transaction's updates are visible to other weak transactions whereas prewrites are visible to all transactions. [LS] presents a new transaction model using isolation-only transactions (IOT). IOTs do not provide failure atomicity and are similar to weak transactions of [PB1].
- An open nested transaction model has been proposed in [C] for modelling mobile transactions as a set of subtransactions. The model allows transactions to be executed on disconnection. It also

supports unilateral commitment of subtransactions and compensating transactions. However, not all the operations are compensated [C], and compensation is costly in mobile computing.

- A kangaroo transaction (KT) model was given in [EH]. It incorporates the property that transactions in a mobile computing hop from a base station to another as the mobile unity moves. The mobility of the transaction model is captured by the use of split transaction [PKH]. A split transaction divides an ongoing transactions into serializable subtransactions. Earlier created subtransaction is committed and the second subtransaction continues execution. The mobile transaction is splitted when a hop occurs. The model captures the data behaviour of the mobile transaction using global and local transactions. The model also relies on compensating transaction in case a transaction aborts. Our model has the option of either using nested transactions or split transactions. However, the save point or split point of a transaction is explictly defined by the use of pre-commit. This feature of the model allows the split point to occur in any of the cell. Unlike KT model, the earlier subtransaction after pre-commit can still continue its execution with the new subtransaction since their commit orders in our model are based on pre-commit point. Unlike KT, our model does not need any compensatry transaction.
- Transaction models for mobile computing that perform updates at mobile computers have been developed in ([CP],[PB1]). These efforts propose a new correctness criterion [C] that are weaker than the serializability. They can cope more efficiently with the restrictions of mobile and wireless communications.
- In ([M1],[M2],[MMCB]), prewrite operations have been used in nested transaction environment to increase concurrency and to avoid undo or compensated operations. The notion of a recovery point subtransaction has been introduced. In a nested transaction tree, if a recovery-point subtransaction executed successfully, its effects are not to be discarded. The crash recovery model has been designed and a correctness proof is carried out using I/O Automaton Model [LM]. In this paper, we exploit some of these ideas in order to increase reliability and availability in mobile computing environment.

3. Prewrite Transaction Model

Formally, the transaction model using prewrite operations [MMCB] has the following features:

- Each prewrite operation makes visible the value the transaction will eventually write. Prewrites have different semantics in different environments. For simple data objects, the prewrite and write values match exactly. For database files, the prewrites may only contain primary-key values and the new values of the fields of records. In case of design objects, prewrites may represent a model of the design. However, the final design released for manufacturing may differ from prewrite design of the model. For example, the final design may have a different colour shade than the prewrite design model. In case of a document object, the prewrite may represent an abstract of the detail document.

- A **pre-read** returns a prewrite value of the data object whereas **read** returns a result of the write operation. A pre-read becomes a **weak-read** (in case the data objects involved are not simple) if it returns a prewrite value even though the transaction who announced the last prewrite has been finally committed. However, this weak-read should not to be aborted. This makes our weak-read different from the weak-read of [PB1].
- Once the required data objects are read or pre-read and prewrites are computed, the transaction pre-commits. A transaction is required to read or pre-read all the required data objects before pre-commit because after a transaction releases a lock for a prewrite operation, it can not get a lock for read operation due to the condition of two phase locking [BHG]. After a pre-commit, the prewrites are made visible to other transactions for processing. Initially, prewrites in our model are kept in the transaction's private workspace. Once the transaction pre-commits, they are posted in the prewrite-buffer. The data objects are physically updated on the disk the pre-commit operations. The prewrites are handled at the transaction manager level and physical writes are handled at the data manager level.
- The transactions commit order in the serilaizable transactions execution history is decided at the order of pre-commit action.
- Our concurrency control algorithm is to be executed in two servers: For controlling pre-read (i.e., read of prewrite value) and prewrite operations at TM server and read (i.e., read of write value) and write operations at DM server. Since prewrite values are made publically visible after pre-commit, the lock-type held by the prewrite operation is converted to the lock-type for write operation after pre-commit. The lock acquired for a prewrite operation is not released after pre-commit because the two phase locking [BHG] does not allow a transaction to acquire a lock after the transaction has released some locks. The concurrent operations and their compatibility-matrix is given in table 1. The locking protocol is given in figure 7.
- A transaction is not allowed to abort after pre-commit. The prewrite provides non-strict execution without cascading aborts. In figure 2, T_1 and T_2 are two subtransactions where pw(x), w(x), pr(x), and r(x) are the prewrite value, write value, pre-read value and read-value respectively, for the data object x. The transaction T_2 commits before T_1. In case T_1 aborts after T_2 commits, there will not be a cascading abort. Since our model does not need "undo recovery" from transaction aborts [MMCB], no compensating transaction is to be executed. In case the pre-committed transaction is forced to abort due to system dependent reasons such as system crash, the transaction restarts on system revival.
- Our model relaxes the **isolation property** as the prewrites are made visible to others after pre-commit but before the final commit of the transaction. Also, durability of prewrites is guaranteed at the pre-commit point.

The formal definitions of a **prewrite operation** and a **pre-committed** transaction are as follows.

Definition 1: A **prewrite operation** announces the value that the data object will have after the commit of the corresponding write operation.

Definition 2: A transaction is called **pre-committed** if it has announced all the prewrites values and read all the required data objects, but the transaction has not been finally committed.

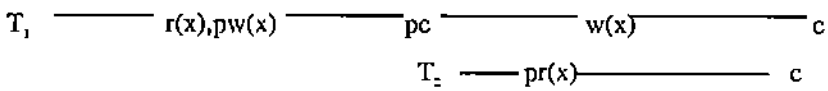

Figure 2. Two Concurrent Transactions

Let us consider two examples that show how a prewrite operation can be used to increase availability.

- Consider the deletion of a record when physical pointers are in use. If the space for the record were de-allocated as a part of deletion, the problems may come if the transaction were to abort. For high concurrency, the storage allocation and de-allocation need to continue once space for the record was de-allocated but before the transaction that de-allocated the space committed. As a result, the space may be allocated to other records, making it impossible for earlier transaction to re-obtain it in case it aborts. In case a new space is allocated for the record, the old pointers may no longer be valid. This problem can be avoided using the prewrite and pre-commit operations. Once the transaction announces the deletion it pre-commits. A pre-committed transaction does not abort, therefore, space can be used by others. The physical deletion of records will be done after pre-commit. Thus, there is no need to acquire the lock on the storage allocator until the transaction commits.
- Consider a construction company which comes up with a 'model-house' for the type of houses it is going to construct in the future. The 'model-house' is shown to prospective customers. The 'model-house' corresponds to prewrite value. Since the 'house-construction' transaction is very long, the pre-view of 'model-house' helps customers in initiating other transactions such as hosing-loan etc. Now, consider a customer who has decided to buy a house. He goes to the bank to arrange a loan. Once the loan request is accepted, the loan has to be uses to buy a house only. The customer has to complete his 'house-buying' transaction. Similarly, the housing company has to construct the number of houses they have sold to customers during pre-view of a 'model-house'.

The formal prewrite transaction processing algorithm is given in figure 3.

---

1. A transaction T is submitted to the system.
2. System analyzer analyzes the transaction T to find out about its read and write requests.
   (/*pre-commit part of the algorithm executed at TM*/)
3. The transaction manager obtains necessary locks on the data objects.

4. If the transaction T has read and write operations or write operations only then

Begin

For all reads ∈ T

Begin

Acquire the necessary locks for reads;

Return values read; (/*prewrite or write values */)

End;

For all writes ∈ T

Begin

Acquire the necessary locks for prewrites;

Announce all the prewrite values;

Store the necessary logs;

End;

Begin (/*execute pre-commit*/)

Update the lock-type of prewrite to lock-type of write provided no other transaction holds conflicting locks;

Write pre-commit log record;

End;

End;

6. For each prewrite announced (/*post pre-commit algorithm executed at DM*/)

Begin

Update those data objects in the database for which prewrites have been announced;

Store necessary log records;

End;

Figure 3. Prewrite Transaction Processing Algorithm

4. Prewrite Mobile Transaction Model

In this section, we see that how prewrites can be used in the mobile transaction applications. We discuss two cases.

Case 1: The MH has limited server capability to do concurrency control, logging, and to execute pre-commit.

In our mobile transaction model, a transaction begins its execution at mobile host. When a transaction arrives at MH, the transaction's read requests are processed at MH in case it has the consistent cached data objects. Otherwise, the MH sends request for some of the reads (for which the MH has no consistent cached data objects) to the MSS. When a read access transaction arrives at MSS, it analyses

the transaction and returns the prewrite value in response. If prewrite is not available, the write values are returned. The MSS tags to identify that the return value is a prewrite value. In case the transaction needs the final write, it has to initiate a read again. Once all the requested values are returned by MSS to MH, the transaction declares all the prewrite values for the data objects and pre-commits at MH. A pre-committed transaction's execution is then shifted from mobile host to the stationary host for the completion of its remaining execution. This moves the expensive part of the mobile transaction execution to the static network. At the stationary host, the transaction actually updates all the data objects for which the prewrite values have been declared earlier and commits thereafter (see figure 4). Thus, the prewrite value of the required data object is made visible by MSS to other mobile hosts in that cell before the data object is updated at the stationary host. This increases availability for concurrently running transactions. Prewrites are stored in the transaction's private workspace at MH and once the transaction pre-commits, they are moved to the main memory of MSS.

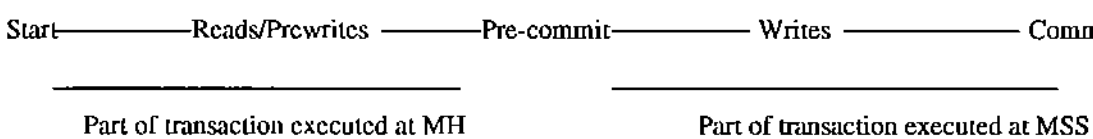

Figure 4. The transaction Processing in Mobile Computing

- Consider a newspaper reporter who is travelling in his van equipped with a mobile computer. If an accident site he likes to report immediately. He initiates a reporting transaction that consists of some 'headline-report' about the accident and sends it to the newspaper office (MSS). The newspaper office immediately displays this on the web page as well as on electronic bulletin board. Since the mobile computer can run out of battery power or due to weak connections or network congestion, the reporter does not prepare and send the full-report. The 'headline-report' corresponds to prewrite value of our model. It reduces the blocking of other transactions at stationary host as it may suffice the requirement of many transactions. For example, based on headline-report, some transactions like reservation of beds in hospitals for victims and transactions for seat reservations in air-lines for the relatives to travel to the accident site etc. can be initiated. Also, once the headline reports arrived at MSS, the reporting transaction's remaining execution is shifted to the MSS for the completion of report. There are two choices for the completion of full-report. First, the newspaper office (MSS) contacts some other reporter in that area to go to the accident site physically and file the complete report. Second, the reporter at MH can also complete the full story at the time of disconnection or weak connections or during lunch-time. Thus, MSS can deal effectively with the situations like failure of MH or in case the mobile host has crossed the cell or is in doze mode. At the time of reconnection, the full-report would be incorporated into the database. The 'headline-report' can also

to be transmitted to other base stations in the fixed network so that transactions there can also be executed without delay.

- **Case 2: The MH acts as I/O only.**

In case the mobile host can not execute the transaction at MH (i.e., MH only acts as I/O), it can submit the transaction to the MSS. The MSS server returns all the required values and declares all the prewrite values corresponding to the write operations. After the transaction pre-commits, the prewrite values are send to the mobile host and at the same time, the rest of the transaction starts executing at the MSS.

- Once a stock-selling transaction is accepted for sell at the given limit-price, the prewrites will have information about the amount of money available before the stock-selling transaction is committed at MSS. Once this information is received at MH, the user can initiate another stock-buying transaction and sends its 'order' to the corresponding MSS.

4.1 Mobile Transaction Model and Partially Replicated System

If the data objects are replicated partially, the mobile transaction makes visible all the prewrites of the required data objects available in its current cell and pre-commits in the current cell. If it does not do this and waits to announce the prewrite values of all other required data objects by moving into different cells, it will block other transactions until it visits those cells. Our approach of pre-committing in the current cell also creates some problems. Once a transaction is pre-committed, it can not again announce prewrite values for the objects in other cells. This is due to the fact that once the prewrite values are made visible at MSS by releasing some locks, it can not again acquire locks due to the two phase locking. To deal with this problem, we use the nested transaction model [M]. Once a transaction has announced all the prewrite values for the data objects available in its current cell and pre-committed, a new subtransaction is created. The earlier subtransaction is serialized before this new subtransaction based on pre-commit point. The earlier subtransaction's execution is shifted to the MSS. The new subtransaction's responsibility is shifted to the MSS of the new cell with the help of hand-off protocol. This solves the problem as each subtransaction is considered a transaction. The pre-commits at various cells are the save points of nested transaction execution.

Another way of handling partially replicated data objects is to split [PKH] the transaction as soon as the MH moves to a new cell. The splitted transaction acts as a new transaction and therefore, can continue its processing in the new cell.

Prewrites can help in partial replication of the data objects in some of the cells it moves. That is, it can make some new servers to support its files. For example, suppose a transaction at MH has send a 'headline-report' about the accident and moved to another cell. In the new cell, it sends the 'headline-report' to its new MSS also. The MSS can therefore, get the headline-report before the full-report is processed at earlier MSS. Thus, the new MSS can serve those mobile transactions, which require headline-report for further processing like reservation of beds in hospitals in the current cell etc. Thus,

transactions at the new MSS are not blocked until the completion of earlier transaction at the previous MSS.

5. Concurrent Operations and Locking

The operation-compatibility matrix of the various operations is given in table 1. In mobile transaction model, a pre-read operation can be executed at both MH and MSS at the same time. A pre-read can be executed at MH (MSS) while a write can take place at MSS (MH) concurrently. Similarly, a read can be executed concurrently with a prewrite. A prewrite operation can also be executed concurrently with another write operation since prewrites are managed at the transaction manager level whereas the writes are performed at the data manager level. However, there are some interesting cases that we discuss below:

- **Case 1:** Suppose a pre-read is currently being executed at MH and at the same time, another transaction who has announced the prewrite values finally commits at MSS (final updates are performed) (see figure 5). In this case, the read will return a prewrite value which might be different than the last write value. If the data object is the simple data object, the read transaction can commit as the prewrite and write values will be same. In case the data is a design object, the system designate the read as a weak- read. The transaction can resubmit its read request later to MSS if it needs the latest complete model of the design.

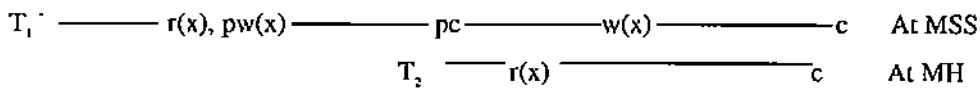

Figure 5. The Situation as discussed in case 1

- **Case 2:** In table 1, observe that a read operation is compatible with a prewrite. Consider a case where a read transaction commits after the other transaction that has completed the prewrite operation and has been pre-committed. The read in this situation will return an old value. However, this is not a significant problem because the transaction can still be serialized (see section 5.1).

	Pre-read	Read	Pre-write	Write
Pre-read	yes	yes	no	Yes
Read	yes	yes	yes	No
Pre-write	no	yes	no	Yes
Write	yes	no	yes	No

Table 1. Operation Compatibility Matrix

We develop a concurrency control algorithm to control the conflicting operations in our mobile transaction model. The concurrency control algorithm is executed in two phases at two places. In the first

phase, the concurrency control for controlling prewrite and pre-read operations is performed at the transaction manager level. In the second phase, the concurrency control for controlling write and read operations (to access write values) is performed at the data manager (DM) level at MSS mainly since the data managers are accessed only while performing updates on the databases.

We use read-lock for read, pre-read-lock for pre-read, prewrite-lock for prewrite, and write-lock for write operations, respectively. A prewrite-lock conflicts with other pre-read- and prewrite-locks, however it does not conflict with read- and write-locks. A prewrite-lock can not be released after pre-commit as the transaction has to still get a write-lock for final updates. A prewrite-lock acquired by a pre-committed transaction is converted to a write-lock provided no other transaction holds the conflicting locks. Once a prewrite-lock is updated to a write-lock, the same transaction can not acquire any other lock [M2]. However, pre-read-locks can be acquired by others to access prewrite values. The locking protocols are given in figure 6. The formal mobile transaction-processing algorithm is given in figure 7.

There will be no a deadlock involving the transactions which are pre-committed. This is due to the fact that prewrite- and write-locks are acquired in an ordered fashion so deadlocks will occur only at the time of acquiring prewrite- or read-locks. Thus, a pre-committed transaction will not be aborted due to the deadlocks.

Locks on the data are managed and provided by the MSS. The number of locks to be acquired in our algorithm depends on the particular replication algorithm used. The two main replication algorithms used are majority consensus and read-one-write all (ROWA) [BHG]. In majority consensus algorithm, locks are acquired on majority of sites whereas read-one-write-all (ROWA) requires lock on all the sites. In case read/write ratio is less, ROWA is preferred otherwise majority consensus will be preferred.

Pre-Read-Lock(X): Grant the requested pre-read-lock to a transaction T on X if no other transaction holds a prewrite-lock on X.

Read-Lock(X): Grant the requested read-lock to a transaction T on X if no other transaction holds a write-lock on X.

Prewrite-Lock(X): Grant the prewrite-lock to a transaction T on X if no other transaction holds a Prewrite- or pre-read-lock on X.

Write-Lock(X): A request to update a prewrite-lock on X held by a transaction T to write-lock on X if

Begin

If the write-lock-wait queue for X is empty **then**

Begin

If the transaction T is pre-committed and no other transaction holds a read- or write-lock on X **then**

convert prewrite-lock to write-lock;

End;

else

Begin

put the transaction T in a write-lock-wait queue for X;

convert the write-lock to prewrite-lock of the transaction at the front of the queue;

End;

End.

Figure 7. The Locking Protocol

---

1. A transaction T is submitted to the mobile host.

2. Mobile host analyzes the transaction T to find out about its read and write requests (we assume here that MH has some server capability).

(/*pre-commit part of the algorithm executed at MH*/)

If the transaction T has read **and** write operations **then**

Begin

For all reads and writes ∈ T

MH sends a request to MSS for all the required read values and request for prewrite-locks;

After MSS acquirs the necessary read-locks, it returns read values to MH (return values will be prewrite value. If no prewrite value is found, the write values are returned

For all writes ∈ T

Begin

Announce all the prewrite values at MH (without waiting for prewrite-locks);

Store the prewrite logs at MH;

Write pre-commit log record and destination move log record;

Send prewrite values, prewrite logs and other log records to MSS;

If MSS acquired all prewrite-locks for all the data objects for which prewrite values have been announced **then**

Begin

MSS accepts the prewrite values and logs;

MSS update the prewrite-lock to write-lock provided no other transaction holds conflicting locks (see figure 7);

End;

else the prewrite values and corresponding logs are discarded at MH;

End;

End;

6. For each prewrite announced (/*post pre-commit algorithm executed at MSS*/)

Begin

Update those data objects in the database for which prewrites have been announced;

Store necessary log records;

End;

Figure 7. Mobile transaction Processing Algorithm

5.1 Correct Schedule in Mobile Transaction Model

In this section, we formalize our mobile transaction model.

Definition 4: A transaction T_i is a partial order with ordering $<_i$ where

1. $T_i \subseteq \{ pr_i(x), r_i(x), pw_i(x), w_i(x) \mid x \text{ is an object} \} \cup \{ pc_i, c_i, a_i \}$.

2. If $a_i \in T_i$ if and only if $pc_i \notin T_i$ and $c_i \notin T_i$.

3. if t is c_i or a_i then for any other operation $p \in T_i$, $p <_i t$ and if t is pc_i, then $pc_i <_i c_i$.

4. if $pr_i(x), r_i(x), pw_i(x), w_i(x) \in T_i$ then either $pr_i(x) <_i w_i(x)$, or $w_i(x) <_i r_i(x)$, or $r_i(x) <_i w_i(x)$, or $pw_i(x) <_i w_i(x)$, $pr_i(x) <_i r_i(x)$, $pw_i(x) <_i pr_i(x)$,

A read operation by a transaction T_i on a data object x is denoted as $r_i(x)$, write as $w_i(x)$, pre-read as $pr_i(x)$ and prewrite as $pw_i(x)$. The pre-commit is denoted as pc_i, commit as c_i and an abort as a_i. The transactions execution history will include lock and unlock operations. Each operation $p_i(x)$ is preceded by a lock operation $pl_i(x)$ and followed by an un-lock operation $pul_i(x)$. Also, $pwl_i(x) \rightarrow wl_i(x)$ implies that a prewrite-lock obtained by transaction T_i on object x is updated to a write-lock.

Now consider five case histories to explain the different situations which might occur during concurrent transactions execution:mmm

Case 1: In this case we consider simple data objects and see that a history with a prewrite is same as the history without prewrite.

Consider the following history:

$$pwl_1(x)pw_1(x)pc_1(pwl_1(x) \rightarrow wl_1(x))prl_2(x)pr_2(x)prul_2(x)c_2w_1(x)wul_1(x)c_1$$

The above history is conflict-serializable in order $T_1 \rightarrow T_2$ and is equal to a serial schedule $T_1 T_2$. However, the above history is not strict since T_2 reads the value from T_1 but commits before T_1. Therefore, it will not be accepted by any scheduler which allows only strict execution as it will introduce cascading aborts. In our algorithm, the transaction T_1 will not abort after its pre-commit operation is executed. Therefore, it is safe to allow non-strict executions in our algorithm. We note that pre-read and

write operations are commutative in case the data objects involved are simple. In this case, a pre-read after a prewrite or a read after its associated final write will return the same value. After taking into account these commutative operations, the above history will be equivalent to:

$$pwl_1(x)pw_1(x)pc_1(pwl_1(x) \rightarrow wl_1(x))w_1(x)wul_1(x)c_1rl_2(x)r_2(x)rul_2(x)c_2$$

This history provides a serial history $T_1\ T_2$. If we remove the prewrite operations from the above history, it will be a serial history consisting of only read and write operations. So we have shown that in the case of simple data objects, any history with prewrites will be equivalent to a history without prewrites. That is, any system which permits pre-read, read, prewrite and write is same as the system with read and write operations in the sense that reads in both the systems will return the same value. However, the system with prewrites permits more concurrency than the system with normal read and writes.

Case 2: In this case we see that once a transaction's prewrite-lock is updated to the write-lock, it can not acquire any other lock.

Consider another history:

$$pwl_1(x)pw_1(x)pc_1(pwl_1(x) \rightarrow wl_1(x))prl_2(x)pr_2(x)pwl_2(y)pw_2(y)pc_2(pwl_2(y) \rightarrow wl_2(y))$$
$$w_2(y)prul_2(x)wul_2(y)c_2w_1(x)rl_1(y)r_1(y)rul_1(y)wul_1(x)c_1$$

The above history is not conflict serializable though it satisfies our locking protocols and two phase locking. The schedule has a cycle since T_1 precedes T_2 on x and T_2 precede T_1 on y. As stated in figure 7, additional restrictions must be introduced to disallow such an execution. Therefore, if a transaction's prewrite-lock is updated to a write-lock then the transaction can not acquire any other lock on any data object.

When this rule is introduced in the above history, T_1 can not acquire the lock on the data object y, therefore, there will not be a cycle.

Case 3: In this case, we see that a prewrite-lock can not be updated to a write-lock if some other transaction is holding a conflicting lock.

Consider a partial history:

$$rl_1(x)r_1(x)pwl_1(x)pw_1(x)rl_2(x)r_2(x)pc_1(pwl_1(x) \rightarrow wl_1(x))$$

The above history is not allowed as T_2 holds a read-lock on x and conflicts with the write-lock acquired by T_1 after pre-commit. Thus, the prewrite-lock is updated to write-lock only if there is no conflicting lock (see write-lock rules in Figure 7).

Consider another partial history:

$$pwl_1(x)pw_1(x)pc_1(pwl_1(x) \rightarrow wl_1(x))plw_2(x)pw_2(x)w_1(x)pc_2(pwl_2(x) \rightarrow w_2(x))$$

The above history is not allowed as T_1 and T_2 now hold conflicting locks. However, if $wul_1(x)$ appears before pc_2 then the history is allowed.

Case 4: In this case, we see that a transaction which returns an old value can be serialized in the history.

Consider another history:

$rl_1(x)r_1(x)pwl_1(x)pw_1(x)rl_2(x)r_2(x)pc_1rul_2(x)c_2(pwl_1(x) \rightarrow wl_1(x))$

The above history is allowed. T_2 returns an old-value since T_1 has pre-committed before T_2 commits. However, this history can be serialized if we move T_2 operations before T_1.

"Our serializabile history satisfies the Q-class [P] of serializable history".

5.2. Proof of Correctness

In this section, we formally prove the correctness of our algorithm and the locking protocols. We prove that the schedule produced by our locking protocols is conflict serializable. Our approach is based on the standard way of proving correctness of transaction processing algorithms.

Here, we state some properties based on our locking protocol.

Property 1 : If o is an operation then ol(x) < o(x) < ou(x).

From two phase locking rule, we have the following property :

Property 2 : If p_i (x) and q_i (y) are two operations under T_i then pl_i (x) < qul_i (y) , i.e., for all lock operations $l_i \in T_i$ and un-lock operations $ul_i \in T_i$, $l_i < ul_i$.

From the lock-upgrading rule, we have the following property :

Property 3 : If $(pwl_i(x) \rightarrow wl_i(x)) \in T_i$ then

(1) for any operation $ol_i \in T_i$, $ol_i(y) <_i (pwl_i(x) \rightarrow wl_i(x))$. That is, once a prewrite-lock is converted to a write -lock, no other operation can lock any data object.

(2) for any operation $pul_i \in T_i$, $(pwl_i(x) \rightarrow wl_i(x)) < pul_i(x)$. That is, a prewrite-lock is converted to a write-lock before any lock is released.

Property 4 : If $p_i(x)$ and $q_j(x)$ conflict then either

1. $pul_i(x) <_H ql_j(x)$. If $p_i(x)$ is a prewrite then $(pwl_i(x) \rightarrow wl_i(x)) <_H ql_j(x)$ or

2. $qul_j(x) <_H pl_i(x)$. If $q_j(x)$ is a prewrite then $(pwl_j(x) \rightarrow wl_j(x)) <_H pl_i(x)$.

Property 5 : If pc_i and c_i are pre-commit and commit then $pc_i <_i c_i$ in T_i and for any pc_j, if $pc_i < pc_j$ then $c_i < c_j$. However, if the transaction T_j is a read-only transaction and $pc_i < pr_j \in$ T_j then $c_i < c_j$ or $c_j < c_i$.

Property 6 : If $pc_i \in T_i$ then no $a_i \in T_i$.

Property 7 : If $pw_i(x)$, $w_i(x) \in T_i$ and $pw_j(x), w_j(x) \in T_j$ and $pw_i(x) < pw_j(x)$ then $w_i(x) < w_j(x)$.

We now state two lemmas before stating the final theorem which proves that any history that satisfies the above properties has an acyclic serialization graph. For a history H, SG(H) has a node for each transaction and an edge $T_i \rightarrow T_j$ if T_i has an operation p_i that conflicts with an operation $q_j \in T_j$.

Lemma 1. If $T_1 \rightarrow T_2$ in SG(H) then there exists an unlock operation $pul_1 \in T_1$ or a lock convert operation $(pwl_1 \rightarrow wl_1)$ such that for all lock operations $ql_2(x)$, $pul_1(x) < ql_2(x)$ or $(pwl_1(x) \rightarrow wl_1(x)) < ql_2(x)$.

Lemma 2. Let $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n$ be a path in SG(H) where n>1. Then for data objects' x and y and some operations $p_1(x)$ and $q_n(y)$ in H, $pul_1(x) < ql_n(y)$ or $(pwl_1(x) \rightarrow wl_1(x)) < ql_n(y)$.

Theorem : Every history H obtained by the locking protocols given before is serializable.

Proof : Suppose by the way of contradiction that SG(H) has a cycle $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$ where n>1. By lemma 2, for some data objects' x and y and some operations $p_1(x)$ and $q_1(y)$ in H, $pul_1(x) < ql_1(y)$ or $(pwl_1(x) \rightarrow wl_1(x)) < ql_1(y)$. But this contradicts property 2. Thus, SG(H) has no cycle and so by the serializability theorem [BHG], H is serializable.

6. How to Deal with System Crash ?

When executing transactions in mobile computing, it is necessary to maintain logs to enable recovery after a system crash. A MH is highly vulnerable to failures due to the loss or theft of equipment, memory loss, etc. In order to recover from such failures, it is necessary to store data objects and their logs at the mobile service stations rather than on mobile host. A MH in our model transfers a transaction's execution to the MSS by moving all the prewrite values and the log records. A separate pre-commit log is maintained for each transaction. Each log record contains the description of the prewrite values. These are appended to the pre-commit record of a transaction. Thus, for every pre-committed transaction, the pre-commit log records for that transaction are stored on the disk as well as in the system log at MSS.

In case of mobile computing, if a MH moves to a different cell, it can also append a record indicating its next possible destination. When the user arrives at the new cell, he can continue the post pre-commit operations if required. Once a transaction's execution is successfully shifted (along with all prewrite logs and pre-commit log) to the MSS, a MH may delete all the log records and keep only prewrite values in main memory for further processing. This way it can gradually discard entries in prewrite logs. To build highly reliable systems, these logs can also be replicated in various cells (MSS) by moving MH. At MSS, prewrite-logs, pre-commit log record, the write logs and final commit log record are stored. If a pre-committed transaction fails at the stationary host, the transaction can be restarted from the point of failure. In case of a system crash, the prewrite logs can be used to build the system state as it existed at the time of pre-commit. Write logs can help to bring the system to the state as it existed at the time of failure with the help of recovery algorithms ([M1],[MMCB]). Since the locks are managed by MSS, the lock tables are maintained at MSS. The transaction table for active transactions are

kept at MSS. However, in case the transaction starts its execution at MH then a part of the transaction table is also maintained at MH until the transaction's execution is shifted to the MSS. Dirty object table is kept at MSS. This table contains information about those objects whose final values are inconsistent with the stable database. This keeps information about those data objects whose prewrite values announced by the pre-committed transactions have not been updated in the database before system crash. In table 2, we give the type of log records and the tables stored at MH and MSS.

In case of a system crash, only the redo of those prewrite and write values will be performed whose effects are not there on stable storage. There is no need for undo pass of the recovery algorithm as data objects are updated only after the transaction is pre-committed.

At MH	At MSS
Prewrite logs	Write log records
Pre-commit log record	Commit log records
'Destination' Move log record	Lock, dirty object and transaction tables

Table 2

7. How does Prewrite Helps in Mobile Computing ?

In this section, we briefly summarise and present some analysis of the following:

- How prewrites in our mobile transaction model help in dealing with constrained resources such as battery power, limited memory, low bandwidth (see figure 8) and frequent disconnection or weak connection.

In mobile computing, to save the energy and to deal with unpredictable failure of mobile host, transaction should be migrated to non-mobile computer in case no further interaction is needed with MH. In case the mobile host transfers the remaining transaction execution (after pre-commit) to the stationary host, it submits all the prewrite values and write operations to the stationary host after pre-commit. The transaction at the stationary host then start updating pre-written data objects and commits. For example, consider a broker who is travelling. While travelling, he would like to place an order for stock purchase. He initiates a buy transaction by submitting the price and the quantity of the stock he would like to buy. He places the order without checking his account as he has to pay only after three working days of purchase. He knows that the payment of his earlier sold stock will arrive during this period. This arrival of "payment" represents a final commit of its earlier transaction. Thus, selling and buying transactions are running concurrently. Since the transaction is very long, placing the order helps in pre-committing the transaction at the MH and remaining execution (buying of stock and payment etc.) is shifted to the MSS. This works well even if the MH is disconnected or it is in sleep state.

Technology	Range	Bandwidth
Ethernet (3Com)	Km	10 Mbps
T1 (ORANet)	Country	1.5 Mbps
2.4 Ghz radio (Proxim)	500ft	1 Mbps
Infrared	30ft	1 Mbps
Satellite (Microspace)	Country	100Kbps
Radio mdm (mobitex)	MAN	10 Kbps
Cellular mdm (Motorola)	MAN	10 Kbps
CDPD (AT&T)	MAN	10 Kbps

Figure 8. Bandwidths of Some Wireless Mediums

- Does prewrite helps in caching data during weak-connections.

A mobile host caches only the prewrite values of the data objects for later processing. For example, a bed reservation transaction at mobile host in section 3 needs the headline report to found about the number of people injured. It does not really need the full story. If the ratio of (size of prewrite-value/size of write-value) becomes smaller then it can be transmitted over low band-width with less power consumption and low-cost. Headline report, for example, will use less memory and battery power, and therefore, can be transmitted over low bandwidth. Similarly, a model of an engineering design can be transmitted faster over low bandwidth rather than the complete design. Consider that if the size of the complete design is 10 Mb and the size of model design is 2 Mb and the bandwidth is 1 Mbps (Infrared, see figure 2). In this case, the network delay on average in case of transmission of complete design from MH to the MSS will be 10 seconds whereas the delay will be 2 seconds in case of transmission of model design. This will decrease the communication cost. Prewrites can be cached during weak connections over low bandwidth due to their small size. This will minimise future network use by MH and improve response time. Furthermore, sending access requests from the mobile host to the MSS may be expensive due to the limited uplink bandwidth and it also uses considerable amount of portable battery power. Thus, by maintaining prewrites at MH can alleviate this problem.

If the user is to be charged based on per time unit based on connection time, the sending or receiving prewrite values will incur less cost than the write values. Similarly, if the user is charged per message basis depending on the length of the message, it will cost less to send or receive prewrite images rather than final design. Since transmitting data consumes more battery power [IB], the sending of the model design will be more cost-effective than sending the completed design. Next, if the read/write ratio becomes greater then prewrite will be more useful in mobile computing environment.

- Does prewrite avoids undo in case of transaction aborts.

Our model **does not need** to handle aborts by using **before-image or compensation**. A pre-committed transaction does not abort. In case a pre-committed transaction is forced to abort due to system dependent

reasons such as crash, the transaction will be restarted on system restart. For example, an accident report transaction need not abort in the case of a failure. In various proposed mobile transaction models, an abort is handled by a compensating transaction. This is costly in mobile computing, as it needs extra resources. In our model, since a transaction's execution is shifted to the MSS after pre-commit, the restart cost will be born by the MSS.

- Does prewrite help in continuing the execution without blocking the transactions and data objects in case of frequent disconnections.

In case the mobile host is disconnected or moved to a different cell, further processing at MH can still continue using prewrites. It need not wait for the commit of earlier transaction at the stationary host. At the same time, stationary host also has the prewrite values, therefore, other transactions at stationary host can continue their execution in case of a disconnection from MH. Since the prewrites are available at MH, it does not require to cache the values again. Thus, in our model, the partial execution of transaction at the mobile host and after pre-commit, the completion of the transaction at the stationary host increases availability with reduced cost.

- Does delayed writes help in reducing network congension.

In our mobile transaction model, final updates are delayed. This action has some advantages in mobile computing. Since write operations are time and resource consuming, they are transferred to MSS for final updates on the database. Delayed writes reduces network traffic congestion caused by updates. The weak-connections in a network can force a mobile host to wait for long in case it would like to pass writes to the servers. Delayed writes also reduce the work-load on distributed file servers since the transactions updates are synchronised at the time of pre-commit.

- Does prewrites simplfy the work of hand-off protocol.

Prewrites reduce the work of hand-off protocol as a pre-committed transaction's execution does not require migration to the new MSS in case MH moves to the new cell. The transaction can still continue at the old MSS due to the fact that the transaction needs no more interaction with MH. If a transaction has not pre-committed and moved to the new cell, it can still continue processing in the new cell and pre-commit there.

8. Conclusion

In this paper, we have presented a new mobile transaction model using prewrites to increase availability. The model allows a transaction's execution to shift from the MH to MSS for final database updates. Thus, reduces the computing expenses. The model needs no "undo" actions or compensating transaction in case of transaction aborts. Prewrite values help in increasing availability as the transactions can be executed during disconnections both at MH and MSS without blocking. We raised some issues to deal with a system failure. For future work, we would like to discuss a detail crash recovery algorithm of our mobile transaction model. We are studying our model in work-flow management, real-time transaction environment and main memory database applications.

References

[BHG] Bernstein P, Hadzilacos, and Goodman, N., Concurrency Control and Recovery in Database Systems, Addison-wesley Publishing Co.,1987.

[BK] Barghouti, N., and Kaiser G., Concurrency Control in Advanced Database Applications, ACM Computing Surveys, 23(3):269-317,1991.

[C] Chrysanthis, P.K., Transaction Processing in a Mobile Computing Environment, Proceedings of IEEE workshop on Advances in Parallel and Distributed Systems, pp.77-82, Oct.1993.

[EH] Eich, M.H.and Helal, A., A Mobile Transaction Model That Captures Both Data and Movement Behaviour, to appear in ACM/Baltzer Journal on Special Topics on Mobile Networks and Applications,1997.

[IB] Imielinksi T. and Badrinath B. R., Wireless Mobile Computing:Challenges in Data Management, Communications of ACM, 37(10), October 1994.

[KS] Kisler J. and M. Satyanarayanan, Disconnected Operation in the Coda File System, ACM Transactions on Computer Systems, 10(1), 1992.

[LM] Lynch, N. and Merrit, M., Introduction to the Theory of Nested Transactions, in *Theoretical Computer Science*, Vol. 62, 1988, pp. 123-185.

[LS] Lu Q. and Satyanaraynan, M., Improving Data Consistency in Mobile Computing Using Isolation-Only Transactions, in proceedings of the fifth workshop on Hot Topics in Operating Systems, Orcas Island, Washington, May 1995.

[M] Moss, J.E.B., Nested Transactions: An Approach to Reliable Distributed Computing, Ph.D. Thesis. Also, Technical Report MIT/LCS/TR-260 MIT Laboratory for Computer Science, Cambridge, MA., April, 1981.

[M1] Madria, S.K., Concurrency Control and Recovery Algorithms in Nested Transaction Environment and Their Proofs of Correctness, Ph.D. Thesis, Department of Mathematics, Indian Institute of Technology, Delhi, 1995.

[M2] Madria, S.K., System Defined Prewrites to Increase Concurrency in Databases, accepted for First East-Europian Symposium on Advances in Databases and Information Systems (sponsored by ACM-SIGMOD), St.-Petersburg (Russia), Sept.97.

[MMCB] Madria, S.K., Maheshwari, S.N, Chandra, B., Bhargava, B., Crash Recovery Algorithm in an Open and Safe Nested Transaction Model, 8th International Conference on Database and Expert System Applications (DEXA), France, Sept.97.

[P] Papadimitriou, C.H., The Serializability of Concurrent Database Updates, J. ACM, 26:4, pp.631-653, 1979.

[PB1] Pitoura E. and B. Bhargava, Building Information Systems for Mobile Environments, Proceedings of 3rd International Conference on Information and Knowledge Management, pp.371-378, 1994.

[PB2] Pitoura E. and B. Bhargava, Maintaining Consistency of Data in Mobile Computing Environments, in proceedings of 15th International Conference on Distributed Computing Systems, June,1995.

[PB3] Pitoura E. and B. Bhargava, Dealing with Mobility: Issues and Research Challenges, Technical Report TR-93-070, Department of Computer Sciences, Purdue University, 1993.

[PKH] Pu C., Kaiser G., and Hutchinson, Split-transactions for Open-ended Activities, in proceedings of the 14th VLDB Conference, 1988.

[RC] Ramamritham K. and Chrysanthis., A Taxonomy of Correctness Criterion in Database Applications, Journal of Very Large Databases, 4(1), Jan.1996.

[WC] Walborn, G. D., Chrysanthis, P.K., Supporting Semantics-Based Transaction Processing in Mobile Database Applications, in proceedings of 14th IEEE Symposium on Reliable Distributed Systems, pp.31-40, Sept.1995.